Contents

1. BOPF Enhancement Workbench Help

Purpose

The BOPF Enhancement Workbench is an application that allows you to enhance existing business objects in a business object processing framework (BOPF). This application help is divided into three sections:

- 1.1 Model Guideline

  This section explains the concepts of enhancing BOPF business objects.

- 1.2 Cookbook

  This section explains where you can use the BOPF Enhancement Workbench.

- 1.3 Implementation Guideline

  This section explains how to implement BOPF entities.

  You can use transaction BOPF_EWB to access the BOPF Enhancement Workbench.

Features

You can use the BOPF Enhancement Workbench to create, change, or delete enhancements of business objects in BOPF, and BOPF enhancements themselves. You can use the following entities to extend business objects:

- Nodes
- Actions and action enhancements
- Determinations
- Consistency and action validations
- Queries

Constraints

You cannot create new business objects with the BOPF Enhancement Workbench. You can only enhance extensible business objects and entities.

1.1 .Model Guideline

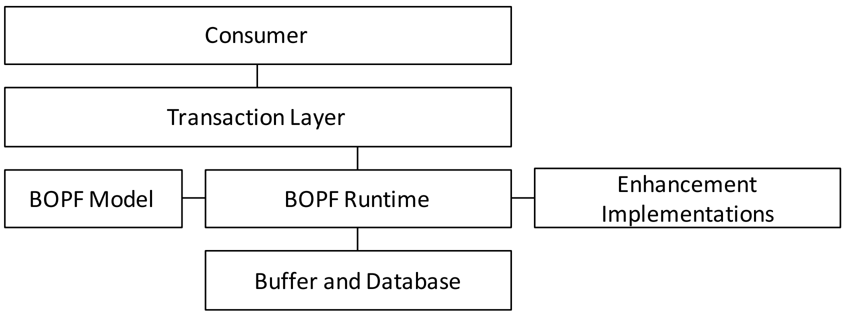
This guideline gives an overview on the following concepts:

- Architecture and transaction model of BOPF
- Business objects and how to extend business objects

The guideline also includes a detailed overview on the entities that build up a business object.

1.1.1 BOPF Architecture

The figure below illustrates the architecture of the BOPF, and is followed by an explanation:



The architecture of the BOPF includes the following five principal areas:

- Consumer

  The consumer uses core services provided by the transaction layer to access business object instances and to control the transaction. Usually the consumer is implemented as a user interface or an autonomous process participant.

- Transaction Layer

  The transaction layer consists of a central transaction manager instance, which allows the consumer to save all changed business object instances of the current transaction. In addition, a service manager instance for each business object provides core services to access all of its business object instances. For example, the core service RETRIEVE allows the consumer to read data from business object instances.

- BOPF Model:

  At design time, each modeled business object consists of several entities. You can maintain this model information in the BOPF Enhancement Workbench. BOPF runtime uses the information. For example, if a certain business object consists of an action entity, the name of its implementing class is stored in this model.

- BOPF Runtime:

  The BOPF runtime executes the requested core services and therefore instantiates and invokes the implementing classes of entities of a business object. For example, if the consumer calls the DO_ACTION core service, the BOPF runtime instantiates and invokes its implementing class and hands over the control. After the action execution, the BOPF collects the result and returns it back to the consumer.

- Buffer and Database:

  To avoid redundant database accesses due to performance reasons, business objects autonomously buffer database accesses. Additionally, the buffer manages the transactional changes.

1.1.2 BOPF Business Object

Definition

A BOPF business object is a representation of a type of uniquely identifiable business entity described by a structural model and an internal process model. Implemented business processes operate on business objects.

Use

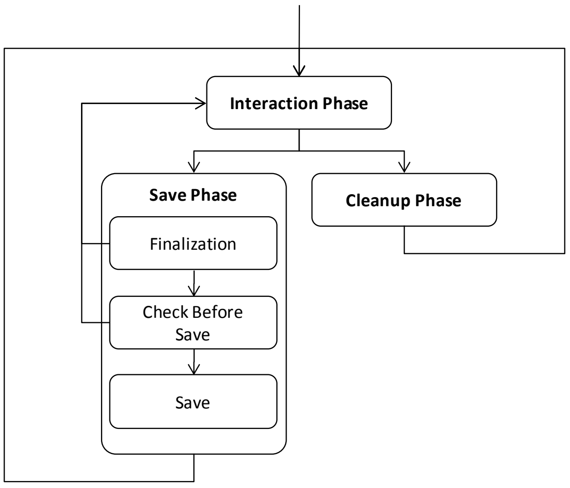
You can use a business object to implement a business object or an application from new. For example, you can use a BOPF business object to implement a complex configuration of a reuse service component.

You can enhance a business object, and its characteristics and configuration settings. A BOPF business object model can consist of all entities described in section 1.1.5.

Example
This document uses the customer invoice business object as an example of a common business object. All invoice data and the corresponding services are encapsulated and provided to the consumer with the help of generic core services.

1.1.3 BOPF Transaction Model

You can divide information processing into indivisible units called transactions. All changes that are done during a transaction must either be cleaned up or saved as a complete unit. A process usually consists of several successive executed transactions. The BOPF transaction model divides the executed transactions into several phases. The figure below illustrates the phases, and is followed by an explanation:



1. Interaction Phase

   At the beginning of the transaction, the consumer can access arbitrary business objects with the help of a set of core services. For example, the consumer can query a certain instance of the customer invoice business object and afterwards execute the action RELEASE.

2. Save Phase:

   As soon as the consumer saves the current transaction, the interaction phase is over and the save phase begins. During this phase, the consumer cannot call core services. The save phase itself is divided into the following phases:

   1. Finalization Phase

      During the finalization phase, each business object participating in the current transaction is prepared for saving. If at least one business object rejects this finalization, the save phase is aborted and the transaction returns to the interaction phase. During the finalization phase, the determinations configured in the pattern *Derive dependent data before saving* are executed. For more information, see 1.1.5.4.2.

   2. Check Before Save Phase

      During this phase, each business object participating in the current transaction is checked to see if it can be saved. If at least one business object fails, the save phase is aborted and the interaction phase starts again. During this phase, all consistency validations with impact *Derive dependent data before saving* are executed. For more information, see section 1.1.5.4.2.

3. Save

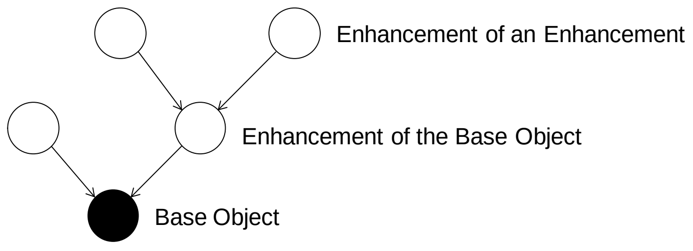
During this phase, the system saves all changes of business object instances in the database.

In addition, the consumer can clean up the current transaction. In this case, the system undoes all changes made to date in the current transaction.

1.1.4 BOPF Business Object Enhancement

Definition
A business object enhancement contains additional business logic and model information that is related to a certain business object or enhancement. We refer to a business object in such a situation as a base object. At design time, each enhancement can be separately maintained in the BOPF Enhancement Workbench. At runtime, the business logic and model information of all enhancements of a business object are merged into one comprehensive object.

Enhancements Hierarchy
At the design time, a base object can be enhanced by more than one enhancement. These enhancements can also have additional enhancements. All enhancements of one base object build up a tree hierarchy. The figure below illustrates a tree hierarchy:



An enhancement can only extend entities that are located in a base object and that are defined as extensible.

Extensible Entities
To extend an entity, it must be defined as extensible.

You can extend nodes with the following additional entities:

- Subnodes
- Attributes
- Determinations
- Actions
- Consistency validation

You can extend actions with the following additional entities:

- Action validations
- Pre action enhancement
- Post action enhancement

1.1.5 Business Object Model

This section describes the following entities of a business object model:

- Nodes
- Associations

- Actions
- Determinations
- Validations
- Queries

1.1.5.1 Nodes

Definition
A node is an entity of a business object that contains data and functionality that are described with the same language. The following are the main features of nodes:

- From a data viewpoint, a node consists of a set of attributes and can be instantiated at runtime. If you compare a node to a table, a node attribute corresponds to one column in the table, and a node instance corresponds to one row.
- If a business object only consists of one node, a business object instance corresponds to one row of the node table. However, usually a business object consists of several nodes whereas each node consists of a set of attributes that are described with the same language.
- The nodes of a business object are linked by the help of associations and build up a hierarchy. The topmost node of the hierarchy is called the root node. Where you have more than one node, a business object instance consists of the following:
  - A row from the table of the root node
  - All rows from the other nodes

    These are linked to the root row with associations.
- From a functional viewpoint, a node is an anchor point for entities that contain business logic related to the data contained in the node.

  Examples include actions operating on the node instances, or queries returning certain node instances with the help of search criteria

  The other core services that are used by the consumer to access a business object instance are always related to a certain node. For example, the retrieve core service returns node instance data for a certain node for a certain business object.

Use
Nodes are used to group attributes of a business object that are described with the same language. These attributes are usually read or written by the consumer at the same time. Dividing sets of attributes into several nodes is essential in order to model quantitative relationships between attributes (for example, 1:N relationship).

Example
The data of a customer invoice can be divided into more than one set that are described with the same language. All data related to the invoice itself is contained in the ROOT node (for example, INVOICE_ID or the RELEASED status of the invoice). The invoice item data is contained in the ITEM node (for example, AMOUNT, ITEM_ID, and QUANTITY). Because one single invoice can consist of more than one item, the ROOT node is linked, using associations, to the ITEM node. In this way, a business object instance of the customer invoice business object consists of one row of the root node table and all rows of the item table that belong to the root row.

There are several entities assigned to the ROOT node of the customer invoice business object. For example, the action RELEASE sets the status of the invoice. The query GET_INVOICE _BY_ID allows you to use the INVOICE_ID to search for invoice instances.

1.1.5.1.1 Persistent Nodes

Definition
The system stores instances of a persistent node in the database when it saves a transaction.

Use
From a database viewpoint, a persistent node corresponds to a table in the database, a node attribute to one database column, and a node instance to one database row.

Example
The CUSTOMER_INVOICE business object consists of several persistent nodes. For example, the ROOT node contains attributes describing global invoice data such as the INVOICE_ID. The persistent node ITEM consists of the attributes AMOUNT, QUANTITY, HEIGHT, WIDTH, and DEPTH.

1.1.5.1.2 Transient Nodes

Definition
The system locates the node instances of a transient node only in the main memory, and does not write transient nodes into the database. Determinations help fill the contents when the consumer accesses the transient node for the first time. For more information about determinations, see 1.2.4.

Use
Transient nodes are useful to buffer data. Buffer data can be derived from persistent fields. This avoids the redundant storage of information in the database and the execution of time consuming deriving functions.

Example
The CUSTOMER_INVOICE business object has a transient subnode called PAYMENT_OPTIONS. Each instance of this node represents one possible payment option. Because the possible payment options for a customer may change, they are not persistent in the invoice business object but dynamically imported and buffered in the transient node during the selection process.

1.1.5.2 Associations

Definition
An association is a direct and unidirectional relationship between two business object nodes. An association can also exist between the same business object nodes.

Use
Associations can be used to relate two nodes in a well-defined direction. You can use an association to navigate from a source node to a related target node. You can define associated nodes in one or more business objects. When you define associated nodes in more than one business object you have cross business object association. You can also use associations to create new node instances of subnodes.

Example
You define the CUSTOMER_INVOICE business object with more than one association. You connect the ROOT node to the ITEM node with an association that has 1:N cardinality.

1.1.5.3 Actions

Definition
An action is an entity assigned to a business object node that describes an operation. You can perform the action on one, more than one, or no instance. An action on no instance is a static action. An action can have an importing parameter, which is handed over by the caller during runtime.

Use
You can use an action to allow the explicit external triggering of business logic. When the system performs the action, the consumer must specify the following:

- The key of the instances on which the action is performed
- Any input parameters that are required by the action

In order to prevent the execution of an action on certain instances, an action validation can be maintained. For more information, see 1.1.5.5.

1.1.5.3.1 Pre Action Enhancement

Definition
A pre action enhancement is an entity of an enhancement that is automatically executed by the framework, before a certain action of the base object is performed. If an importing parameter structure is maintained on the base action, this parameter is also handed over to the pre action enhancement.

Use
A pre action enhancement can be used to extend the functionality of a certain action that is located in the base object.

Example
The CUSTOMER_INVOICE business object contains the action RELEASE to release an invoice. Before an invoice can be released, it must be approved by the action INVOICE_APPROVE. This action sets the APPROVAL attribute of the invoice. To provide a customer solution that automatically approves invoices with a total amount lower than USD 1.000, the pre action enhancement AUTO_APPROVAL is created in an enhancement. It runs before the RELEASE action is executed and sets the APPROVAL attribute in accordance with the invoice total amount.

Restrictions
- A pre action enhancement can only be created in a business object enhancement and must be related to a base action located in the base business object of the enhancement.

- If the execution of the base action on this instance is prevented by an action validation, this instance is not processed by the pre action or the base action.

- A pre action enhancement cannot return failed instances.

  All instances are also processed by the base action.

1.1.5.3.2 Post Action Enhancement

Definition
A post action enhancement is an entity of an enhancement that is automatically executed by the framework after a certain action of the base business object is performed. If an importing parameter structure is maintained on the base action, this parameter is also handed over to the post action enhancement.

Use
A post action enhancement can be used to extend the functionality of a certain action that is located in the base business object. The instances passed to the post action enhancements have already been successfully processed by the base action.

Example
The CUSTOMER_INOVICE business object contains the action INVOICE_ISSUED that releases a certain invoice. A pre action enhancement assigned to the enhancement ENH_CUSTOMER_INVOICE is added to inform the manager of the accountant in case invoices have a total amount greater than EUR 100.000.

Restrictions

- If the execution of the base action on this instance is prevented by an action validation, this instance is not processed by the base action or its post action enhancements.

- If the instance fails the processing of the base action, it is not handed over to the post action enhancements. The post action enhancements of a base action can only operate on instances that are successfully processed by the base action. If the base action execution fails on all instances, no post action enhancement is executed.

1.1.5.4 Determinations

Definition

A determination is an element assigned to a business object node that describes internal changing business logic on the business object. Like a database trigger, a determination is automatically executed by the BOPF as soon as the BOPF triggering condition is fulfilled. This triggering condition is checked by the framework at different points in the transaction, depending on the pattern of the determination. For each determination, it is necessary to specify the changes that build the triggering condition. Changes can include creating, updating, deleting, or loading node instances.

Use

A determination is mainly used to compute data that can be derived from the values of other attributes. The determined attribute and the determining attributes of the triggering condition can belong to the same node or to different nodes. There are also values that do not depend on any other value but still have to be determined automatically on the creation or modification of a node instance, for example IDs.

Determination Dependencies

As soon as the framework checks the trigger conditions of determinations and there is more than one determination to be executed, the dependencies of the determinations are considered. With the help of a determination dependency, a determination can be maintained either as a predecessor or a successor of another determination.

Determination Dependencies Example

After changing the quantity of invoice items, the triggering conditions of the determinations CALCULATE_ITEM_AMOUNT and CALCULATE_TOTAL_AMOUNT are both fulfilled. CALCULATE_ITEM_AMOUNT calculates the amount of the changed item (price x quantity), CALCULATE_TOTAL_AMOUNT sums the amounts of all items to the total amount of the invoice.

If you don't maintain any dependencies, the system could calculate the total amount before the item amount. Therefore, you must maintain the CALCULATE_ITEM_AMOUNT determination as a predecessor of the CALCULATE_TOTAL_AMOUNT determination.

Determination Patterns

Depending on the use case, the framework checks the triggering condition of a determination at several points during the transaction.

1.1.5.4.1 Pattern "Derive dependent data immediately after modification"

Definition

The trigger condition of a determination of the pattern "Derive dependent data immediately after modification" is evaluated at the end of each modification. A modification roundtrip is defined as one single modification core service call from the consumer to the framework. The call contains arbitrary creations, updates, or deletions of node instances. Additionally, the trigger condition is checked after each action core service execution.

Use
This pattern shall be used if the creation, the update, or the deletion of node instances causes unforeseen errors. These errors are handled during the same roundtrip.

If there is no need to react immediately on the modification, and the handling of the side effect is very time consuming, we recommend you use the *Derive dependent data before saving* determination pattern instead.

Example
As soon as a new ITEM node instance of the CUSTOMER_INVOICE business object is added, the changed item amount (price x quantity) must be immediately recalculated in order to show the new amount on the consumer's user interface.

1.1.5.4.2 Pattern "Derive dependent data before saving"

Definition
The trigger condition of determinations configured to this pattern is checked as soon as the consumer saves the whole transaction. If the save of the transaction fails, these determinations could run multiple times.

Use
In contrast to the "Derive dependent data immediately after modification" pattern, the framework evaluates all changes done so far in the current transaction to check the trigger condition. Because this evaluation only takes place at the save phase of the transaction, this pattern is recommended for time consuming determinations.

Example
The data of each invoice must additionally be stored in a XML file. A determination is configured to extract the XML code from each changed invoice. Because this is very time consuming the determination does not run immediately after each change of an invoice. Instead, it runs once before saving for all invoices changed during the current transaction.

1.1.5.4.3 Pattern "Fill transient attributes of persistent nodes"

Definition
Determinations configured in this pattern are automatically executed before the consumer accesses a transient node attribute of the assigned node for the first time. This allows you to initially derive the values of the attribute. In addition, these determinations are executed after each modification of a node instance. This allows you to recalculate the transient field if its derivation source attribute has been changed by the modification.

Use
These determinations are used to derive the values of transient attributes of a node.

Example
The volume of a certain item of a customer invoice can be derived out of the length, width, and depth of this item unit, and the quantity. The volume attribute is a transient attribute of the item node, and its value can be derived as soon as an item is loaded from the database. Therefore, you can use a determination that calculates and fills the volume at this point in time .

1.1.5.4.4 Pattern "Derive instances of transient nodes"

Definition
Determinations of this pattern are executed before the consumer accesses the assigned transient node of the determination and allows the creation, update or deletion of transient node instances.

Use

These determinations are used to create and update instances of transient nodes. Because the determinations are executed before each access to their assigned transient node, they must ensure that the requested instances are in a consistent state.

Example

The customer invoice business object does not store payment options. The payment option node is a transient node buffering detail information that is located in another system.

1.1.5.5 Validations

Definition

A validation is an element of a business object node that describes some internal checking business logic on the business object.

Use

A validation can be used to either validate if a certain action can be executed on certain node instances (action validation) or if a set of node instances is consistent (consistency validation).

1.1.5.5.1 Action Validations

Definition

An action validation is a business object node entity which is referred to a certain action. It contains checking logic which is automatically executed before the action is processed.

Use

Action validations can be used to check if an action can be carried out. An action validation is carried out when an action is called, and before it is performed. If some validations fail, the action is not performed for the instances where the validation failed.

Example

With the help of the RELEASE action of the CUSTOMER_INVOICE, business object invoices can be set to status released. Before an invoice is allowed to be released, it must be approved. If a consumer calls the action RELEASE, the CHECK_RELEASE action validation is automatically executed. It checks the instances on which the release action has been called. Only instances having the APPROVED flag set are processed by the RELEASE action.

1.1.5.5.2 Consistency Validations

Definition

A consistency validation is a business object node entity that can be used to check the consistency of the nodes instances. In contrast to actions and determinations, it just contains read-only logic. Each consistency validation configuration consists of a triggering condition which is checked by the framework at several time points during the transaction. If the triggering condition is fulfilled, the consistency validation is executed. If you have inconsistent node instances, a consistency validation reacts as if its impact is maintained as follows:

- Sending messages to the consumer, updating on the inconsistency details
- Sending messages to the consumer and preventing the saving of the transaction until the inconsistency is corrected.
- Sending messages to the consumer and changing a consistency status variable.

  A consistency status variable indicates the status of a node instance.

Use
Consistency validations can be used to check the consistency of a business object. It is possible to check whether or not a certain set of node instances of a certain node are consistent. The consistency validation implementation returns a set of failed keys identifying all handed over node instances that are inconsistent.

Example
The CUSTOMER_INVOICE business object contains several consistency validations to check if the items of an invoice are in a consistent state.

For example, one consistency validation checks if the quantity and price of each item is maintained. Another consistency validation checks if the address of the buyer is valid. If not, the consistency validation prevents the saving of the transaction to ensure that only consistent invoice instances can be saved.

1.1.5.6 Queries

Definition
A query is a business object node entity that returns the queried keys and data of the business object node instances. To allow different search criteria at runtime, the consumer may hand over the query data type.

Use
Queries provide the initial point of access to business objects. They allow you to perform searches on a business object to receive the keys or the data of certain or all node instances. Each query has an associated parameter structure. The result of the query is a set of all the record IDs in a business object that match the query criteria.

1.1.5.6.1 Node Attribute Query

Definition
A node attribute query is a type of query where the search parameters are equal to the assigned node of the query. The search criteria can consist of value comparisons and value ranges on the attributes of the nodes. The query returns the instances of the assigned node whose attributes match the search parameters that have been handed over.

Use
Node attribute queries are recommended for all cases that do not need complex query logic. In contrast to custom queries, node attribute queries are only modeled and therefore do not have to be implemented.

Example
A node attribute query named SELECT_INVOICE is assigned to the ROOT node of the CUSTOMER_INVOICE business object. This allows the consumer to search invoices by the help of the attributes of the ROOT node, e.g. the INVOICE_NUMBER 48775444 or the CREATION_DATE 2009-04-12.

Restrictions
- The query data type corresponds to the combined structure of the assigned node.

  The search criteria are always reduced to the attributes of the assigned node.

- The query only returns instance data or keys of node instances for its assigned node.

- The query returns the node instances of the assigned node where the attribute values fit the search criteria.

1.1.5.6.2 Custom Query

Definition

A custom query is a type of a query that executes application specific logic. In contrast to the node attribute query, this logic must be implemented in the implementing class of the query. The custom query can have an arbitrary data type structure that is handed over by the consumer to the query implementation at runtime. In addition, the implementation is able to return an arbitrary data table as the result of the query. The result data table must contain a key component relating to the instances of the assigned node.

Use

Custom queries must be used if the recommended node attribute queries do not fulfill the requirements. This is the case if specific query parameters must be handed over, or if the query logic is more complex than comparing attribute values.

Example

The customer invoice business object allows you to search for expired dates. The GET_EXPIRED_INVOICES query is assigned to the ROOT node.

Restrictions

The query only returns instance data or keys of the node instances of its assigned node or an arbitrary data table.

1.2 BOPF Enhancement Workbench Cookbook

This chapter explains how you use the BOPF Enhancement Workbench in certain important use cases. We recommend that you read the BOPF Model Guideline before using this document to get an understanding of the main concepts behind the workbench.

1.2.1 Enhancements

1.2.1.1 Create and Change an Enhancement

The first step in adding functionality to an existing business object or enhancement is to create a new enhancement. The BOPF Enhancement Workbench provides you with a wizard that is described in this chapter. This wizard also allows you to change an existing enhancement.

Starting the Wizard

Precondition

Only business objects and enhancements that are configured as extensible can be extended by a new enhancement.

Procedure

To start the wizard, in the *Business Object Browser*, select the business object or enhancement that you need to extend with a new enhancement. In the context menu, choose the *Create Enhancement* pushbutton. You can change an enhancement that already exists using the context menu entry *Change Enhancement*.

Maintain the Name, Prefix, and Namespace

Procedure

- Enhancement Name

  Enter a meaningful name for the new enhancement.

- Namespace

The ABAP namespace avoids naming conflicts of the enhancement and its entities. If no namespace is maintained, you must add a prefix. The namespace can have no more than 10 characters.

- Description

  Enter a short description about the purpose of the enhancement.

- Prefix

  The prefix allows you to distinguish between the entities of several business objects that belong to the same namespace. If no prefix is maintained, you must enter a namespace of less than 10 characters.

Restrictions

- The prefix and the namespace combined must consist of less than 10 characters.

Maintain the Constants Interface

Procedure

- *Technical Name*

  The technical name is automatically derived from the name, namespace, and prefix of the previously entered enhancement. This name is displayed later in the *Business Object Browser* of the enhancement workbench.

- *Constants Interface*

  This field contains the constants interface name. This interface will be created automatically by the wizard and contains the human-readable names of all the entities contained in the enhancement.

Maintain the Extensibility of the Enhancement

In this step, you must maintain the extensibility of the enhancement.

Procedure

If the enhancement is not marked as extensible, you cannot create further enhancements based on this enhancement.

Finishing the Wizard

Procedure

To create the enhancement, choose the *Complete* pushbutton. The wizard performs all changes outlined below. If you choose the *Cancel* pushbutton, the enhancement is not created and the wizard terminates without making any changes.

Result

- The system creates and displays the enhancement in the *Enhancement Browser* when you select the corresponding assigned node.
- The system generates the constants interface maintained in this enhancement if the interface does not already exist.

1.2.1.2 Delete an Enhancement

Procedure

To delete an enhancement, in the context menu of the enhancement browser, choose *Delete Enhancement*. The delete wizard also allows you to remove the constants interface of the enhancement.

Constraints

- Do not delete an enhancement that has already been transported.

The enhancement can be in use in a follow-up system.

- You must delete all entities of the enhancement before you can delete the enhancement.

  For example, all nodes belonging to a particular enhancement, and all actions, determinations, and validations must be deleted first.

1.2.2 Subnodes

1.2.2.1 Create and Change a Subnode

It is possible to extend a business object with additional nodes. You can add subnodes to an existing enhancement of a particular business object with the help of a wizard. This wizard also allows you to change an already existing enhancement subnode.

Starting the Wizard

Precondition

You can only assign subnodes to the following node types:

- Nodes that are created in the enhancement that is currently open

  The system marks these in blue in the browser menu.

- Extensible nodes that belong to a base object

  The system marks these in black in the browser menu.

Procedure

In the browser menu, choose the node that you need to extend with a new subnode. In the context menu, choose *Create Subnode* to start the wizard.

In the context menu of the node, choose *Change Subnode* to change a node that already exists.

Maintain Name and Description

Procedure

- *Parent Node Name*

  This is the parent node to which the new subnode is assigned in the model of the business object.

- *Subnode Name*

  The name of the subnode must be unique in the business object and should start with the namespace of the open enhancement. If no namespace has been entered, the node name must start with the prefix of the open enhancement. This ensures you have clear separation between the nodes of different enhancements that belong to the same business object. The namespace (or prefix) value is automatically inserted in this field and must be completed with a meaningful node name.

- *Description*

  Enter a short description about the purpose of the subnode.

Maintain Extensibility and Extension Includes

In this step, you must specify whether or not the subnode is extensible in further enhancements.

Procedure

If you need to make the subnode extensible with the following in further enhancements, select the *Extensible* checkbox:

- Additional determinations
- Consistency validations
- Actions
- Action validations
- Queries
- Subnodes

If you need to make the subnode extensible with supplementary node attributes in further enhancements, you have the option of using the *Persistent Extension Include* or *Transient Extension Include* features. Enter a name in the *Persistent Extension Include* field or copy the naming proposals and create the structure by double-clicking the name.

Restriction
You can only assign the structure defined in the *Persistent Extension Include* and *Transient Extension Include* input fields to the enhancement category *Character-Type, Numeric,* or *Deep*.

Maintain Attributes
Each node consists of several attributes. There are two types of attribute, as follows:

- Persistent attributes

  The system stores these in the database

- Transient attributes

  These only hold in memory. You must maintain the structures that contain these attributes.

Procedure
For both the *Persistent Structure* and *Transient Structure* fields, enter the name of the structure that contains the persistent attributes of the subnode or copy the naming proposal of the wizard. Later, you can create the structure by double-clicking the name. You must activate the structure to continue the wizard. You can only choose the enhancement categories *Can be enhanced (character-type or numeric)* or *Can Be Enhanced (Deep)*.

Restriction
If you update just the *Transient Structur*e, the node is called a transient node. Subnodes of transient nodes must only have transient attributes, and no persistent attributes. Therefore for a subnode that is assigned to a transient parent node, the *Persistent Structure* input field is automatically hidden in this screen.

If you have already defined a *Persistent Extension Include* you must specify it in the *Persistent Structure* as an inclusion. This same condition also applies to *Transient Extension Include*.

Example
The ITEM_DETAILS subnode in the ITEM parent node contains more information about items. The persistent attributes are DEPTH, WIDTH, HEIGHT, and COLOR. The attribute CAPACITY is a transient attribute, because it can be calculated out of DEPTH, WIDTH, and HEIGHT at runtime.

Maintain Combined Structure, Combined Table Type, and the Database Table

Procedure
All entities of this screen can be automatically created by the wizard. Therefore we recommend you continue without changing the proposed names.

- *Combined Structure*

The *Combined Structure* combines the *Transient Structure*, the *Persistent Structure,* and a key include in order to identify each node instance with the help of a key. All attributes of the subnode are represented by the structures components.

- *Combined Table Type*

  The combined table type consists of lines of the combined structure.

- *Database Table Name*

  The database table contains the data of the persistent attributes of all subnode instances.

Finishing the Wizard

You have entered all relevant information about the new subnode.

Procedure

To create the subnode, choose the *Complete* pushbutton. The wizard performs all the changes below. If you choose the *Cancel* pushbutton, the system does not create the subnode and the wizard terminates without making any changes.

Result

- The system adds the new subnode to the enhancement model and displays it in the menu browser of the node.
- The system generates the *Combined Structure* and the *Combined Table Type*.

  For a persistent subnode, the system also creates the *Database Table*. Already existing entities are not overwritten.

- The constants interface of the enhancement is regenerated and contains a unique constant identifying the subnode.

  This constant is necessary in order to access the data of the node.

1.2.2.2 Delete a Subnode

Procedure

To delete a subnode, open the enhancement browser. In the context menu, choose *Delete Enhancement*. You can also remove the transient and persistent extension include, and the data structures of a node. In addition, you can delete the combined structure and the database table for a persistent node. The deletion of the database also removes the data of the node instances.

Constraints

- Do not delete a subnode that has already been transported.

  It could already be in use in one of the follow-up systems.

- You must delete all entities of the enhancement assigned to the node before you can delete the node.

  For example, you must first delete all actions, determinations, and validations belonging to a node in the enhancement.

1.2.3 Actions

1.2.3.1 Create and Change an Action

An action is an element of a business object node that describes an operation performed on that node. For more information, see 1.1.5.3

In contrast to consistency validation or determination, it can be directly triggered by the consumer. This section explains how the wizard creates and changes enhancement actions.

Starting the Wizard

Precondition

You can only assign actions to the following two types of nodes:

- Nodes that are created in the current open enhancement

  The system marks these nodes in blue in the entity browser.

- Extensible nodes belonging to a base object.

  The system marks these nodes in black in the entity browser.

Procedure

In the *Node Browser*, choose the node to which you need to assign the action. In the context menu, choose *Create Action* to start the wizard. To change an enhancement action that already exists, in the context menu of the enhancement action, chose *Change Action*.

Maintain Name and Description

Procedure

- *Node Name*

  This displays the node to which the action is assigned in the model of the business object.

- *Action Name*

  The action name should start with the namespace or prefix of the open enhancement. This ensures there is a clear separation between the entities of different enhancements. The system automatically enters the value in this field. You should add a meaningful action name.

- *Description*

  Enter a short description about the purpose of the action.

Maintain Implementing Class, Cardinality, and Parameter Structure

Procedure

- *Implementing Class*

  The implementing class contains the business logic of the action. The system creates it automatically after finishing the wizard. You must implement it manually.

  Because the implementing class name should meet naming conventions, the wizard automatically suggests a valid class name. You can also define a class that already exists, by implementing the /BOBF/IF_FRW_ACTION interface as the implementing class. This is useful if the action logic of another action can be reused. The system does not overwrite the implementing class if it already exists.

- *Action Cardinality*

  This defines how many node instances the action can operate on during one action call. The following are the action cardinality types:

  - *Multiple Node Instances*

    Select if the action always operates on one or more node instances.

  - *Single Node Instance*

    Select if the action operates on exactly one single node instance for each call.

  - *Static Action (No Node Instances)*

    Select if the action does not operate on any node instances.

- *Parameter Structure*

  Some actions need an additional importing parameter. Enter a name for the parameter structure and create the structure by double-clicking the name.

Finishing the Wizard

Procedure

To create the action, choose the *Complete* pushbutton. The wizard performs all changes outlined below. If you choose the *Cancel* pushbutton, the system does not create the action and the wizard terminates without making any changes.

Result

- The system adds the new action to the enhancement model

  It displays the action in the *Entity Browser* when you select the corresponding assigned node.

- The implementing class of the action is generated and must be implemented.
- The constants interface of the enhancement is regenerated and contains a unique constant identifying this action.

  The consumer needs the constant to execute the action.

1.2.3.2 Delete an Action

Procedure

To delete an action open the assigned node in the node browser. In the context menu of the action, choose *Delete Enhancement*. The delete wizard also allows you to remove the implementing class and the parameter structure of the action.

Constraints

- Do not delete an action that has already been transported.

  The action could already be in use in a follow-up system. For instance, it can be enhanced by a pre action enhancement or post action enhancement.

- All entities of the enhancement assigned to this action must be deleted before the action can be deleted.

  For example, all of its action validations, pre action enhancements, and post action enhancements must be deleted before the action can be deleted.

1.2.3.3 Create and Change a Pre Action Enhancement

A *pre action enhancement* is an action enhancement that is automatically executed before its related action is executed. This chapter explains how to create a new pre action enhancement for a particular action. This wizard can also be used to change an existing pre action enhancement.

Start the Wizard

Precondition

Only extensible actions for a base object can be enhanced with pre action enhancements.

Procedure

In the *Entity Browser*, choose the action which will be enhanced by the new pre action enhancement. To start the wizard, in the context menu choose *Create Pre Action Enhancement*. To change a pre action enhancement that already exists, in the context menu of the existing pre action enhancement, choose *Change Pre Action Enhancement*

Maintain Name and Description

Procedure

- *Node Name*

  This field displays the assigned node to which the determination will be assigned in the business object's model.

- *Base Action Name*

  The base action field contains the name of the action that will be extended by the pre action enhancement.

- *Action Name*

  The pre action enhancement's name should start with the namespace or prefix of the open enhancement. This value is automatically inserted in this field and must be continued with a meaningful pre action enhancement name.

- *Description*

  Enter a short description about the purpose of the pre action enhancement.

Maintain the Implementing Class

Procedure

The *Implementing Class* contains the business logic of the pre action enhancement. It is automatically created after finishing the wizard. You must subsequently implement it manually. Because the implementing class name should meet naming conventions, the wizard automatically suggests a valid class name. You can also define a class that already exists, by implementing the /BOBF/IF_FRW_ACTION interface as the implementing class. This is useful if the action logic of another action can be reused. The system does not overwrite the implementing class if it already exists.

Finishing the Wizard

Procedure

To create the pre action enhancement, choose the *Complete* pushbutton. The wizard performs all changes outlined below. If you choose the *Cancel* pushbutton, the system does not create the pre action enhancement and the wizard terminates without making any changes.

Result

- The system adds the new pre action enhancement to the enhancement model

It displays the pre action enhancement in the *Entity Browser* when you select the corresponding assigned node.

- The implementing class of the pre action enhancement is generated and must be implemented.
- The constants interface of the pre action enhancement is regenerated.

1.2.3.4 Delete a Pre Action Enhancement

Procedure
To delete a pre action enhancement open the assigned node in the node browser. In the context menu of the action, choose *Delete Pre Action Enhancement*. The delete wizard also allows you to remove the implementing class of the pre action enhancement.

Constraints
Do not delete an action that has already been transported. The action could already be in use in a follow-up system.

1.2.3.5 Create and Change a Post Action Enhancement

A post action enhancement is an action enhancement that is automatically executed after its related action is executed. For more information, see section 1.1.5.3.2. This chapter explains how to create or change a post action enhancement.

Start the Wizard

Precondition
The following types of actions can be extended by a post action enhancement:

- Actions that are created in the *Enhancement* that is currently open.

  In the *Entity Browser*, these actions are marked in blue.

- Extensible actions that belong to a base object.

  In the *Entity Browser*, these actions are marked in black.

Procedure
In the *Entity Browser*, choose the action which will be enhanced by the new post action enhancement. To start the wizard, in the context menu choose *Create Post Action Enhancement*. To change a post action enhancement that already exists, in the context menu of the existing post action enhancement, choose *Change Post Action Enhancement.*

Maintain Name and Description

Procedure

- *Node Name*

  This field displays the assigned node to which the post action enhancement will be assigned in the model of the business object.

- *Action Name*

  The action field contains the name of the base action that will be extended by the post action enhancement.

- *Action Name*

The name of the post action enhancement should start with the namespace or prefix of the opened enhancement. This value is automatically inserted in this field and must be continued with a meaningful post action enhancement name.

- *Description*

  Enter a short description about the purpose of the post action enhancement.

Maintain the Implementing Class

Procedure

The *Implementing Class* contains the business logic of the post action enhancement. It is automatically created after finishing the wizard. You must subsequently implement it manually. Because the implementing class name should meet naming conventions, the wizard automatically suggests a valid class name. You can also define a class that already exists, by implementing the /BOBF/IF_FRW_ACTION interface as the implementing class. This is useful if the action logic of another action can be reused. The system does not overwrite the implementing class if it already exists.

Finishing the Wizard

Procedure

To create the post action enhancement, choose the *Complete* pushbutton. The wizard performs all changes outlined below. If you choose the *Cancel* pushbutton, the system does not create the post action enhancement and the wizard terminates without making any changes.

Result

- The system adds the new post action enhancement to the enhancement model

  It displays the post action enhancement in the *Entity Browser* when you select the corresponding assigned node.

- The implementing class of the post action enhancement is generated and must be implemented.
- The constants interface of the enhancement is regenerated.

1.2.3.6 Delete a Post Action Enhancement

Procedure

To delete a post action enhancement, open the assigned node in the node browser. In the context menu of the post action enhancement, choose *Delete Post Action Enhancement*. The delete wizard also allows you to remove the implementing class of the post action enhancement.

Constraints

Do not delete an already transported post action enhancement. It could be in use in one of the follow-up systems.

1.2.4 Determinations

1.2.4.1 Create and Change a Determination

A determination is an entity of a business object that allows you to handle unforeseen errors. For more information, see section 1.1.5.4. This chapter explains how to create or change a determination.

Starting the Wizard

Precondition

You can only assign determinations to the following types of nodes:

- Nodes that are created in the current open enhancement

  The system marks these nodes in blue in the entity browser.

- Extensible nodes belonging to a base object.

  The system marks these nodes in black in the entity browser.

Procedure

In the *Node Browser*, choose the node to which you need to assign the determination. In the context menu, choose *Create Determination* to start the wizard. To change a determination that already exists, in the context menu of the enhancement action, chose *Change Determination*.

Maintain Name and Description

Procedure

- *Node Name*

  This field displays the assigned node to which the determination will be assigned in the model of the business object.

- *Determination Name*

  The determination's name should start with the namespace or prefix of the open enhancement. The value is automatically inserted in this field and must be continued with a meaningful determination name.

- *Description*

  Enter a short description about the purpose of the determination.

Maintain the Implementing Class

Procedure

The *Implementing Class* contains the business logic of the determination. The system creates it automatically after finishing the wizard. You must implement it manually.

Because the implementing class name should meet naming conventions, the wizard automatically suggests a valid class name. You can also define a class that already exists, by implementing the /BOBF/IF_FRW_DETERMINATION interface as the implementing class. This is useful if the action logic of another action can be reused. The system does not overwrite the implementing class if it already exists.

Maintain the Determination Pattern

Procedure

Depending on the selected pattern, the triggering condition of the determination is checked by the framework at different points in time during the current transaction. For more information, see 1.1.5.4.

Precondition

- The *Fill transient attributes of persistent nodes* pattern can only be selected if the assigned node also contains transient attributes.

- The *Derive Instances of transient nodes* pattern can only be selected if a transient subnode of the assigned node of the determination exists.

Maintain Request Nodes

Precondition
The *Maintain Request Nodes* screen only appears if you choose the *Derive dependent data immediately after modification* or *Derive dependent data before saving* determination patterns.

Procedure
You can maintain the triggering condition of the determination. The displayed nodes are linked via associations to the assigned node of the determination. You can define for each node if the creation, update, or deletion of one of its instances will trigger the execution of the determination.

Example
The determination CALCULATE_ITEM_AMOUNT shall run as soon as a new ITEM node instance is created or is updated, for instance the quantity or the price attribute. In both cases the amount must be calculated.

Restriction
At least one request node must be selected in order to ensure that the determination is executed.

Maintain Transient Node

Precondition
The *Maintain Transient Node* screen appears only if you choose the *Derive Instances of Transient Nodes* determination pattern.

Procedure
Select the transient subnode whose instances the determination modifies.

Maintain Write Nodes

Precondition
This screen only appears if there is more than one locking shadow defined in the base object.

Procedure
Select all nodes whose instances are created or modified by the determination.

Maintain Determination Dependencies

Precondition
This screen only appears if there is more than one determination configured to the same determination pattern. This can only be the case for the *Derive dependent data immediately after modification* and *Derive dependent data before saving* patterns.

Procedure
Select the determinations which must be processed before or after the current determination. The triggering condition of the determination must also be fulfilled.

Restriction
Determinations of enhancements are always executed after determinations of the base object. This means that dependencies can only be maintained between determinations belonging to the enhancement.

Example
The determination calculate_total_amount sums all item amounts to the total invoice amount. Therefore it must be executed after determination calculate_item_amount, which calculates the item amounts (quantity x price).

Finishing the Wizard

Procedure
To create the determination, choose the *Complete* pushbutton. The wizard performs all changes outlined below. If you choose the *Cancel* pushbutton, the determination will not be created and the wizard terminates without making any changes.

Result
- The system adds the new determination to the enhancement model and displays it in the *Node Browser*.
- The implementing class is generated.

  You must implement it manually.

- The constants interface of the enhancement is regenerated and contains a unique constant identifying this determination.

1.2.4.2 Delete a Determination

Procedure
To delete a determination, in the node browser open the assigned node. In the context menu of the determination, choose *Delete Determination*. The delete wizard starts and also allows you to remove the implementing class of the determination.

1.2.5 Validations

1.2.5.1 Create and Change a Consistency Validation
This chapter explains how to create an additional consistency validation in an action that already exists.

Starting the Wizard

Precondition
You can only assign consistency validations to the following two types of nodes:

- Nodes that are created in the current open enhancement

  The system marks these nodes in black in the entity browser.

- Extensible nodes belonging to a base object.

  The system marks these nodes in blue in the entity browser.

Procedure
In the node browser, choose the node to which you need to assign the consistency validation. In the context menu, choose *Create Consistency Validation* to start the wizard. Only instances of this node can be checked by the consistency validation.

Maintain Name and Description

Procedure
- *Node Name*

  This displays the assigned node to which the consistency validation will be assigned in the model of the business object.

- *Validation Name*

The name of the consistency validation should start with the namespace or prefix of the open enhancement. This ensures a clear separation between the consistency validations of different enhancements that belong to the same business object. The value is automatically inserted in this field and must be continued with a meaningful node name.

- *Description*

  Enter a short description about the purpose of the node.

Maintain Implementing Class

Procedure

The implementing class contains the business logic of the consistency validation. The system creates it automatically after finishing the wizard. You must then manually implement it.

Because the implementing class name should meet naming conventions, the wizard automatically suggests a valid class name. You can also define a class that already exists, by implementing the /BOBF/IF_FRW_VALIDATION interface as the implementing class. This is useful if the action logic of another consistency validation can be reused. The system does not overwrite the implementing class if it already exists.

Maintain Request Nodes

A consistency validation is automatically executed as soon as one of the triggering conditions of its request nodes is fulfilled. For more information, see 1.1.5.5. At this wizard step, the request nodes and the corresponding triggering condition must be defined.

Procedure

The *Maintain Request Nodes* screen shows all nodes that are connected to the assigned node by an association. To maintain a request node, select the request node checkbox and the appropriate triggering condition (create, update, or delete).

Maintain Impact

The purpose of a consistency validation is to indicate changed node instances that are inconsistent by the help of messages. You can prevent the system from saving the entire transaction if a changed instance fails the consistency validation, or fails to set a consistency status. You can maintain the type of reaction on inconsistent node instances as validation impacts in this screen.

Procedure

- *Return messages*

  This represents the default behavior of consistency validation. The validation implementation returns messages for inconsistent instances to the consumer.

- *Return messages and prevent saving*

  If the inconsistency of a node instance must be solved before saving the transaction, this validation impact is selected.

- *Return messages and set a consistency status*

  If a base object contains a consistency status, this status can be influenced by a consistency validation. Choose the appropriate status variable. When a changed instance fails the consistency validation, this status is automatically set to inconsistent.

Finishing the Wizard

Procedure
To create the consistency validation, choose the *Complete* pushbutton. The wizard performs all changes outlined below. If you choose the *Cancel* pushbutton, the system does not create the consistency validation and the wizard terminates without making any changes.

Result
- The system adds the new consistency validation to the enhancement model and displays it in the *Entity Browser*.
- The implementing class of the action is generated and must be implemented.
- The constants interface of the enhancement is regenerated and contains a unique constant identifying this consistent validation.

1.2.5.2 Create and Change an Action Validation
Action validations can prevent the execution of an action. For more information, see 1.1.5.5. This section explains how to create an additional action validation in an action that already exists with the help of a wizard.

Start the Wizard

Precondition
You can only extend the following two types of actions with an action validation

- Actions that are created in the current open enhancement

  The system marks these nodes in black in the entity browser.

- Extensible actions belonging to a base object.

  The system marks these nodes in blue in the entity browser.

Procedure
In the *Business Object Browser*, choose the business object or enhancement that will be extended by a new enhancement. In the context menu, choose *Create Action Validation*.

Maintain Name and Description

Procedure
- *Action Name*

  This displays the name of the action whose execution is either allowed or prevented by the action validation at runtime.

- *Validation Name*

  The validation name of the action should start with the namespace or prefix of the open enhancement. This value is automatically inserted and must be continued with a meaningful *action validation* name.

- *Description*

  Enter a short *description* about the purpose of the action validation.

Maintain the Implementing Class

Procedure
The implementing class contains the business logic of the action validation. The system creates it automatically after finishing the wizard. You must implement it manually.

Because the implementing class name should meet naming conventions, the wizard automatically suggests a valid class name. You can also define a class that already exists, by implementing the /BOBF/IF_FRW_VALIDATION as the implementing class. This is useful if the logic of another action validation can be reused. The system does not overwrite the implementing class if it already exists.

Finishing the Wizard

Procedure

In order to create the Action Validation, choose the *Complete* button. The wizard performs all changes mentioned below. If you choose the *Cancel* pushbutton, the system does not create the action validation and the wizard terminates without making any changes.

Result

- The system adds the new action validation to the enhancement model and displays it in the *Node Browser*.
- The implementing class is generated and must be manually implemented.
- The constants interface of the enhancement is regenerated and now contains a unique constant identifying this action validation.

1.2.5.3 Delete a Validation

Procedure

To delete a validation, open the assigned node in the node browser. In the context menu of the validation, choose *Delete Action Validation* or *Consistency Validation*. The delete wizard starts and allows you to additionally removing the implementing class of the validation.

1.2.6 Query

1.2.6.1 Create and Change a Query

With the help of this wizard, you can create new queries for an enhancement. For more information about the query concept and the usage of queries see section 1.1.5.6.

Starting the Wizard

Precondition

You can only assign queries to the following types of nodes:

- Nodes that are created in the current open enhancement

  The system marks these nodes in blue in the node browser.

- Extensible nodes belonging to a base object.

  The system marks these nodes in black in the node browser.

Procedure

In the node browser, choose the node to which the query will be assigned. In the context menu, choose *Create Query* to start this wizard. The result of the query only contains a subset of node instances of this assigned node. For example, it is not possible to assign a query to the ROOT node, which returns instances or their keys of the ITEM node.

Maintain Name and Description

Procedure

- *Node Name*

This displays the node to which the query will be assigned in the model of the business object.

- *Query Name*

  The query name should start with the namespace or prefix of the open enhancement. This ensures there is a clear separation between the entities of different enhancements. The value is automatically inserted in this field and must be continued with a meaningful query name.

- Description

  Enter a short description about the purpose of the query.

Maintain the Query Type

Procedure

Queries can be distinguished into node attribute queries and custom queries, as follows.

- The advantage of the node attribute query is that their logic is fully provided from the framework and therefore they must not be implemented.

  However, the set of query input parameters corresponds to the set of attributes of the assigned node. The input parameters are resolved similar to common range selections. For more information, see 1.1.5.6.

- Use custom queries to implement the entire query logic and to freely define the input parameters of the query.

Maintain Query Configuration

Precondition

This wizard step only appears if the query type custom query has been selected in the previous screen. In contrast to custom queries, node attribute queries must not be implemented.

Procedure

- *Implementing Class*

  The implementing class contains the business logic of the query. The system creates it automatically after finishing the wizard. You must implement it manually.

  Because the implementing class name should meet naming conventions, the wizard automatically suggests a valid class name. You can also define a class that already exists, by implementing the /BOBF/IF_FRW_QUERY interface as the implementing class. This is useful if the query logic of another query can be reused. The system does not overwrite the implementing class if it already exists.

- *Data Type*

  If the consumer of the query is able to hand over dynamic query input parameters at runtime, the data type of the query must be specified. Technically, it is a structure and each of its components corresponds to one single selection criteria. For example, if the query returns instances that cost more than a certain amount, the structure must contain the component AMOUNT.

- *Result Type (optional)*

  The result type is an arbitrary data type which can be used as the result of an custom query. For example to provide a denormalized read only list of BO data for the UI. This configuration is optional.

- *Result Table Type (optional)*

A result table which has the result type as the row type.

Finishing the Wizard

Procedure
To create the query, choose the *Complete* pushbutton. The wizard performs all changes outlined below. If you choose the *Cancel* pushbutton, the system does not create the query and the wizard terminates without making any changes.

Result

- The system adds the new query to the enhancement model.

  When you select the corresponding assigned node in the *Node Browser,* it displays this query in the *Entity Browser.*

- The constants interface of the enhancement is regenerated and contains a unique constant identifying this query.

  The consumer needs this constant to execute the query.

- The system only creates the implementing class if you have chosen to create a custom query.

  The system does not overwrite implementing class that already exists.

1.2.6.2 Delete a Query

Procedure
To delete a query, in the node browser open the assigned node. In the context menu of the query choose *Delete Query*. The delete wizard starts and allows you to remove the implementing class of the query, the data type, the result type, and the result table type.

Constraints
Do not delete a transported query that already exists. It could be in use in one of the follow-up systems.

1.3 BOPF Implementation Guideline

This guideline gives you information on the implementation of BOPF entities. It is divided into the following sections:

- Consumer implementation

  This section describes how to implement a BOPF consumer.

- Implementation of entities

  This section describes how to implement BOPF entities such as determinations, validations, and actions.

1.3.1 Consumer Implementation

Consumer implementation consists of the following three sections:

- Core services

  Introduces the core services, which allow the consumer to access business object instances.

- Transaction services

Describes the transaction services SAVE and CLEANUP, which save the changes done by the core services.

- Consumer message handling

  Handles the messages returned by the core services and transaction.

1.3.1.1 Core Services

The consumer (for example, the user interface or a foreign business object) is allowed to access a business object with the help of a set of core services. These core services are described in detail in the interface documentation of the service manager /BOBF/IF_TRA_SERVICE_MGR.

- CHECK_ACTION:

  This method allows you to check if a certain action can be executed. You have the option of using a given action parameter on a given set of node instances. The execution of an action can be prohibited depending on the state of the node instances, the properties of the action, or failed action validations.

- CHECK_AND_DETERMINE

  With this method, you can check whether or not a certain set of node instances of a node is currently in a consistent state. The validations configured to the corresponding node and the relevant determinations are executed in order to make the business object instance consistent. Additionally, the consistency statuses covered by the check scope are updated.

- CHECK_CONSISTENCY

  This method checks if a certain set of node instances of a model node is in a consistent state. The validations configured to the corresponding node or of the complete substructure are executed in order to check the consistency.

- CONVERT_ALTERN_KEY

  This method retrieves for certain node instances that are identified by an alternative key (called source alternative key) the corresponding alternative key values of another alternative key on the same node (called target alternative key).

- DO_ACTION

  This method executes a certain action with a given parameter on a given set of node instances.

- MODIFY

  This method allows you to create, update, and delete node instances in a mass enabled manner.

- RETRIEVE

  The retrieve method allows you to read data of certain instances of a certain node.

- RETRIEVE_BY_ASSOCIATION

  This method allows you to retrieve node instances that can be identified with the help of a certain associat

- GET_NEW_KEY

  This method returns a new globally unique key, which has not been used to identify any other entities. For example, this can be used to identify new node instances.

- QUERY

  This method executes a query and returns the query result.

The following core services are only relevant for user interfaces accessing BOPF business objects.

- RETRIEVE_CODE_VALUE_SET

  A value set is the BOPF representation of a dynamic value help (F4 help) of an entity. This method can be used to receive the values for a certain code list.

- RETRIEVE_DEFAULT_ACTION_PARAM

  This method allows you to retrieve the dynamic default parameter values of a certain action. Due to the fact, that the selection of the default parameter of an action can depend on the instances on which the action shall operate as well as a prefilled action parameter, IT_KEY and CS_PARAMETERS are import parameters of this function. The method returns a reference to a parameter structure of that action, which is enriched by the default values.

- RETRIEVE_DEFAULT_NODE_VALUES

  This method can be used in order to fetch dynamic, context dependent default values for node instances, without actually creating them.

- RETRIEVE_DEFAULT_QUERY_PARAM

  This method can be called to receive the default parameter values of a certain query. Due to the fact, that the default parameter of a query can depend on a prefilled parameter, the CT_SELECTION_PARAMETER parameter is imported.

- RETRIEVE_PROPERTY

  This method allows you to retrieve the properties of certain entities.

Restrictions
The core services offered by the service manager must not be used out of entity implementations in order to access the individual business object. The READ object provided by the framework must be used instead.

1.3.1.2 Transaction Services

Procedure
The transaction model is divided into an interaction phase and a save phase. For more information, see 1.1.3. After finishing the core service calls, a consumer can either cleanup or save the changes done in the current transaction. In both cases, the transaction manager instance must be received from the factory class /BOBF/CL_TRA_TRANS_MGR_FACTORY=>GET_TRANSACTION_MANAGER. Afterwards, the corresponding instance method CLEANUP or SAVE can be called.

Restrictions
The transaction services offered by the transaction manager must not be called in entity implementations. Only the consumer is allowed to control the transaction.

Example
The following example shows how to save a transaction after the interaction phase took place.

```
DATA lo_transaction_manager TYPE REF TO /bobf/if_tra_transaction_mgr.
" start interaction phase
…
" start save phase
```

```
lo_transaction_manager = /bobf/cl_tra_trans_mgr_factory=>get_transaction_manager( ).
lo_transaction_manager->save( ).
```

1.3.1.3 Consumer Message Handling

Most of the core services return a message object (EO_MESSAGE) back to the consumer, which contains messages created by the executed BOPF entities. In order to make use of them, the following attributes of messages must apply:

- Depending on the lifetime, the validity period of the message is either restricted to the current service call (lifetime transaction) or it continues until the inconsistency is corrected (lifetime state).
- The severity of the message is expressed by the message severity. It is distinguished into error, warning, and information.
- The origin location defines which instance the message is referred to.

1.3.2 Implementation of Entities

This section is about the implementation of BOPF entities. After describing common concepts, the implementation of the following entities is explained in detail:

- Common implementations
- Actions
- Determinations
- Validations
- Query

1.3.2.1 Common Implementations

This section is about the following common concepts in implementing BOPF entities.

- Constants interface
- Message and change handling
- Reading access
- Modifying access
- Entity context
- Accessing foreign business objects

1.3.2.2 Constants Interface

Each entity of a business object or an enhancement can be identified by a unique model key. This technical key is automatically generated as soon as the entity has been created. All keys of the modeled entities are represented with the help of constants in the constants interface. These constants are named equal to their corresponding entities. This interface is automatically created and updated. The constants must be used as parameter for certain core services.

The constants interface is structured as follows:

Entity	Constant and Constant Schema
Actions	SC_ACTION SC_ACTION-<NODE_NAME>-<ACTION_NAME>

Action Parameter Attributes	SC_ACTION_ATTRIBUTE SC_ACTION-<NODE_NAME>-<ACTION_NAME>-<ATTRIBUTE_NAME>
Alternative Keys	SC_ALTERNATIVE_KEY SC_ALTERNATIVE_KEY-<NODE_NAME>-<ALTERNATIVE_KEY_NAME>
Associations	SC_ASSOCIATION SC_ASSOCIATION-<SOURCE_NODE_NAME>-<ASSOCIATION_NAME>
Association Attributes	SC_ASSOCIATION_ATTRIBUTE SC_ASSOCIATION-<SOURCE_NODE_NAME>-<ASSOCIATION_NAME>-<ASSOCIATION_ATTRIBUTE_NAME>
Business Object (or Enhancement)	SC_BO_KEY
Name of the Business Object (or Enhancement)	SC_BO_NAME
Determinations	SC_DETERMINATION SC_DETERMINATION-<NODE_NAME>-<DETERMINATION_NAME>
Version of the Model	SC_MODEL_VERSION
Nodes	SC_NODE SC_NODES-<NODE_NAME>
Attributes of Nodes	SC_NODE_ATTRIBUTE SC_NODE_ATTRIBUTE-<NODE_NAME>-<NODE_ATTRIBUTE_NAME>
Node Categories	SC_NODE_CATEGORY SC_NODE_CATEGORY-<NODE_NAME>-<NODE_CATEGORY_NAME>
Queries	SC_QUERY SC_QUERY-<NODE_NAME>-<QUERY_NAME>
Attribute of the Queries	SC_QUERY_ATTRIBUTE SC_QUERY_ATTRIBUTE-<NODE_NAME>-<QUERY_NAME>-<QUERY_ATTRIBUTE_NAME>
Validations	SC_VALIDATION SC_VALIDATION-<NODE_NAME>-<VALIDATION_NAME>

Example

The action INVOICE_ISSUED assigned to the ROOT node is technically represented with the help of key 48288AEE887B50FCE10000000A42172F. To increase the readability of coding only the constant (zif_cons_interf_cust=>sc_action-root-invoice_issued) must be used in the implementation. To retrieve instances of the ITEM node, the RETRIEVE core service parameter IV_NODE_KEY is set to zif_cons_interf_Cust=>sc_node-item.

1.3.2.3 Message and Change Handling

The BOPF supports the use of T100 messages, which can be created by business object entities in order to be forwarded to the consumer. For example, a consistency validation should return an error message for an inconsistent node instance. From an implementation viewpoint, a message is an instance of a class, which inherits the BOPF exception class /BOBF/CM_FRW. A message contains additional context information in which the following attributes must be handed over to the constructor.

- TEXTID

  This component is written as SYMSG and the MSGID and MSGNO must be filled with the message number and message ID. This is done with the help of a constant of the corresponding CM class.

- SEVERITY

  There are different severities that classify a message as an error, a warning, or an information message. To specify the message severity, use the constants of the /BOBF/CM_FRW class.

- LIFETIME

  The validity period of this message depends on its lifetime, and can be automatically set. Use the constant /BOBF/IF_FRW_C=>SC_LIFETIME_SET_BY_BOPF.

- MS_ORIGIN_LOCATION

  This attribute must be filled with the key of the node instance, which is affected by the message.

You implement message handling as follows:

1. Clear the exporting parameter EO_MESSAGE.
2. Create an instance of the suitable BOPF message class and hand over the information described above.
3. Get the current message container from the BOPF factory (/BOBF/CL_FRW_FACTORY=>GET_MESSAGE).
4. Add the message to the message container.

Example

```
" clear message object at the beginning
DATA:
  ls_location  TYPE        /bobf/s_frw_location,
  lo_cm        TYPE REF TO cm_bopf_training.
CLEAR eo_message.
…
" create message instance
ls_location-node_key = is_ctx-node_key.
ls_location-key      = ls_key-key.
CREATE OBJECT lo_cm
  EXPORTING
    textid             = cm_bopf_training =>missing_bill_to_party
    severity           = /bobf/cm_frw=>co_severity_error
    lifetime           = /bobf/if_frw_c=>sc_lifetime_set_by_bopf
    symptom            = /bobf/if_frw_message_symptoms=>co_bo_inconsistency
    ms_origin_location = ls_location.
…
" get message container
IF eo_message IS NOT BOUND.
  CALL METHOD /bopf/cl_frw_factory=>get_message
    RECEIVING
      eo_message = eo_message.
ENDIF.
…
" add message to the message container
eo_message->add_cm( lo_cm ).
```

1.3.2.4 Reading Access
When implementing the business logic of BOPF entities, you may need to read node instance data. For example, a consistency validation must read the node instance data corresponding to the node instance keys that have been handed over, to check certain attributes. The BOPF supports reading access of business object instances with the help of an access object, which is provided as importing parameter. It implements the /BOBF/IF_FRW_READ interface, which provides the following methods:

- RETRIEVE

  Returns the target node instance data of the requested node instances.

- RETRIEVE_BY_ASSOCIATION

  Follows an association to retrieve node keys and data.

- CONVERT_ALTERN_KEY

  Allows you to convert one (alternative) key into another (alternative) key.

- GET_ROOT_KEY

  Returns the keys of the root node instances of the node instance keys that have been handed over.

- COMPARE

  Compares the current and previous image of node instances IT_KEY of node IV_NODE. The result is passed as a change object. If required (IV_FILL_ATTRIBUTES = true) the attributes of the node instances are also compared and the result is written into the change object. The scope defines If just the node is specified or if the complete sub-tree is compared (only supported on ROOT nodes and not for delegated objects).

More Information
For more information about these methods. see the interface documentation of the /BOBF/IF_FRW_READ interface.

1.3.2.5 Modifying Access
Determinations and actions modify node instance data.

The BOPF provides an access object, which is provided as an importing parameter. It implements the /BOBF/IF_FRW_MODIFY interface, which provides the following methods.

- CREATE

  Creates a new instance of a node.

- UPDATE

  Updates a single instance of a node.

- DELETE

  Deletes one or many node instances.

- DO_ACTION

  Executes a certain action.

- DO_MODIFY

Allows the creation, update, or deletion of a set of node instances by a single call.

- END_MODIFY

  After calling this method, the modifications that were made during the processing of the calling methods are immediately executed on the buffer. You only need to trigger the modifications explicitly if processing the content method relies on the values contained in the parameters EO_MESSAGE and EO_CHANGE.

- NOTIFY_CHANGE

  This is a deprecated method; do not use.

- NOTIFY_ASSOCIATION_CHANGE

  Marks outgoing associations of a node as changed.

- NOTIFY_PROPERTY_CHANGE

  Creates a property change notification and passes it to the service consumer.

More Information
For more information about these methods, see the interface documentation of the /BOBF/IF_FRW_MODIFY interface.

1.3.2.6 Entity Context
You can use model information in the implementation of an entity. The BOPF provides an entity context as an importing parameter of the interface of the entity. It contains information about the business object, the assigned node of the entity, the key of the entity, and further static model information.

Example
In order to ensure that an invoice amount is only entered in USD or EUR, the validation CHECK_CURRENCY is maintained at the ROOT and the ITEM node. Because it is identical validation logic, the same implementing class is maintained for both consistency validations. The retrieve core service must be called on the assigned node of the validation whose node key can be received from the validation context.

1.3.2.7 Accessing Foreign Business Objects
In order to access instances of foreign business objects, the service manager instance of your business object must be instantiated. Afterwards, all core services provided by the service manager interface /BOBF/IF_TRA_SERVICE_MGR can be used. For more information about the services, see the interface documentation.

Example
```
" start of interaction phase
DATA lo_service_manager TYPE REF TO /bobf/if_tra_service_mgr.
lo_service_manager = /bobf/cl_tra_serv_mgr_factory=>get_instance( iv_bo_key = if_ci_bopf_customer_invoi_c=>sc_bo_key ).
lo_service_manager->retrieve(…).
```

1.3.2.8 Admin Data
In many scenarios it is essential to keep track of the date, time, and causes of the creation or the last change of a node instance. The BOPF supports admin data with the help of special determinations, which store the information in special attributes. If a node that is affected by one of those determinations is extended with the BOPF Enhancement Workbench, the extended instances of the subnode are automatically taken into account.

Example
The customer invoice business object consists of an ITEM node. As soon as one of its instances is changed or created, the user and the changing time is stored in the ITEM attributes CREATED_BY,

CREATION_TIME, CHANGED_BY, and CHANGE_TIME. If the ITEM node is enhanced by a subnode, these attributes are overwritten as soon as one instance of this subnode is created or changed.

1.3.2.9 Locking

The BOPF automatically locks node instances as soon as they are modified by the implementation class of entities. Because of this, no locking logic has to be implemented.

1.3.2.10 Actions

To create an action, the implementing class of the action must be implemented. This class corresponds to a common ABAP class, with the /BOBF/IF_FRW_ACTION interface implemented. If the action is created by the BOPF Enhancement Workbench, the generated class already implements this interface. For more information on the BOPF Enhancement Workbench, see 1.1.5.3.

The action interface consists of the following three methods:

- PREPARE

  This optional method allows you to enlarge or reduce the set of keys for which the action is executed. This method is processed before the validations of the action and the execute method of the action are processed.

- EXECUTE

  This mandatory method contains the business logic of the action.

- RETRIEVE_DEFAULT_PARAM

  If the action has an importing parameter maintained, it can be useful for the consumer to get a default parameter structure. In order to provide the consumer a default parameter, this optional method must be implemented.

More Information

For more information on the model guidelines on actions, see 1.1.5.3. For more information on how to create, change, and delete actions, see 1.2.3.

Example

The following source code describes the action INVOICE_PAID, which can be used by the consumer in order to mark invoices as paid:

```
METHOD /bobf/if_frw_action~execute.
 DATA:
   lv_ltimestamp TYPE          timestampl,
   lt_root_data  TYPE          bobf_t_customer_invoice,
   ls_root_data  TYPE REF TO           bobf_s_customer_invoice.

 " clear exporting parameter
 CLEAR:
   eo_message
   et_failed_key.
 " get all root instance data
 io_read->retrieve(
   EXPORTING
     iv_node = if_ci_bopf_customer_invoi_c=>sc_node-root
     it_key  = it_key
   IMPORTING
     et_data = lt_root_data.
 " set the attributes PAYED and PAYMENT_RECEIVED
 LOOP AT lt_root_data REFERENCE INTO ls_root_data.
   " update data
   ls_root_data->payed = if_sp_pay_exec_st_cd=>co_received.
   " set payment date time stamp
   GET TIME STAMP FIELD lv_ltimestamp.
   ls_root_data->payment_received = lv_ltimestamp.
   " update data
   io_modify->update(
     EXPORTING
```

```
        iv_node = if_ci_bopf_customer_invoi_c=>sc_node-root
        iv_key  = ls_root_data->key
        is_data = ls_root_data ).
  ENDLOOP.
ENDMETHOD.
```

1.3.2.11 Determinations

In order to create a determination, the implementing class of the determination must be implemented. This class corresponds to a common ABAP class with the interface /BOBF/IF_FRW_DETERMINATION maintained. If the determination is created by the BOPF Enhancement Workbench, the generated class already implements this interface. For more information on the BOPF Enhancement Workbench, see 1.1.5.4.

The determination interface /BOBF/IF_FRW_DETERMINATION consists of the following methods:

- CHECK_DELTA

  This optional method checks the relevance of node instances for the execution of the determination, based on changes made since the last determination run (if there was a run) at the field level.

- CHECK

  This optional method checks the relevance of node instances for the execution of the determination based on field values. This method does not have access to a before state and is therefore not capable of checking whether there were relevant changes on node instances.

- EXECUTE

  This method executes the determination and can modify node data.

More Information

For more information on the model guidelines on determinations, see 1.1.5.4. For more information on how to create, change, and delete determinations in the BOPF Enhancement Workbench, see 1.2.3.

Example

The following source code describes the determination ITEM_AMOUNT. It recalculates the item amount whenever the quantity or the price of that item is changed.

```
METHOD /bobf/if_frw_determination~execute.
  DATA:
    lt_item_data      TYPE        bobf_t_ci_item.
    ls_item_data      TYPE REF TO bobf_s_ci_item,

  " clear exporting parameter
  CLEAR:
    eo_message
    et_failed_key.
   " read the instance data of the item
  io_read->retrieve(
    EXPORTING
      iv_node = if_ci_bopf_customer_invoi_c=>sc_node-item
      it_key  = it_key
    IMPORTING
      et_data = lt_item_data ).
  " loop over all instances
  LOOP AT lt_item_data REFERENCE INTO ls_item_data.
    " calculate new amount
    ls_item_data->net_amount = ls_item_data->quantity * ls_item_data->price_amount.
    " update data
    io_modify->update(
      EXPORTING
        iv_node = is_ctx-node_key
        iv_key  = ls_item_data->key
        is_data = ls_item_data ).
  ENDLOOP.
ENDMETHOD.
```

1.3.2.12 Validations

To create a validation, the implementing class of the validation must be implemented. This class corresponds to a common ABAP class, with the /BOBF/IF_FRW_VALIDATION interface maintained. If the validation is created by the BOPF Enhancement Workbench, the generated class already implements this interface. For more information about the BOPF Enhancement Workbench, see 1.1.5.5,

The validation interface /BOBF/IF_FRW_VALIDATION consists of the following methods:

- CHECK_DELTA

  This optional method checks the relevance of node instances for the execution of the validation, based on changes made since the last validation run (if there was a run) at the field level. For action validations, this method is not executed because changes to action validations are made before this point in time and are therefore not relevant.

- CHECK

  This optional method checks the relevance of node instances for the execution of the validation based on field values. This method does not have access to a before state and is therefore not capable of checking whether any relevant changes were made on node instances.

- EXECUTE

  This mandatory method executes the validation. It has no direct modifying access to the node data but returns a set of failed keys to identify all inconsistent node instances.

More Information

For more information on the model guidelines on validations, see 1.1.5.5. For more information on how to create, change, and delete validations in the BOPF Enhancement Workbench, see 1.2.5.

Example

The following source code describes an action validation. The action validation ensures that only node instances that have a valid business partner instance maintained can be processed by the INVOICE_ISSUED action.

```
METHOD /bobf/if_frw_validation~execute.
  DATA:
    ls_key       TYPE              /bobf/s_frw_key,
    ls_location  TYPE               /bobf/s_frw_location,
    lt_key_link  TYPE                    /bobf/t_frw_key_link,
    lo_cm        TYPE REF TO         cm_bopf_training.

  " clear exporting parameter
  CLEAR:
    et_failed_key,
    eo_message.
  " get the party node instances of the IT_KEY root instances
  io_read->retrieve_by_association(
    EXPORTING
      iv_node        = if_ci_bopf_customer_invoi_c=>sc_node-root
      it_key         = it_key
      iv_association = if_ci_bopf_customer_invoi_c=>sc_association-root-party
      iv_fill_data   = abap_false
    IMPORTING
      et_key_link    = lt_key_link ).
  LOOP AT it_key INTO ls_key.
    READ TABLE lt_key_link with KEY source_key = ls_key-key TRANSPORTING NO FIELDS.
    IF sy-subrc <> 0.
    " this node instance has no corresponding party instance
      INSERT ls_key INTO TABLE et_failed_key.
      CLEAR ls_location.
      ls_location-node_key = is_ctx-node_key.
      ls_location-key      = ls_key-key.
      CREATE OBJECT lo_cm
        EXPORTING
          textid           = cm_bopf_training=>missing_bill_to_party
          severity          = /bobf/cm_frw=>co_severity_error
          lifetime          = /bobf/if_frw_c=>sc_lifetime_set_by_bopf
          symptom           = /bobf/if_frw_message_symptoms=>co_bo_inconsistency
          ms_origin_location = ls_location.
      IF eo_message IS NOT BOUND.
        CALL METHOD /bopf/cl_frw_factory=>get_message
          RECEIVING
            eo_message = eo_message.
      ENDIF.
      eo_message->add_cm( lo_cm ).
    ENDIF.
  ENDLOOP.
ENDMETHOD.
```

1.3.2.13 Queries

1.3.2.13.1 Node Attribute Query

Node attribute queries cannot be implemented. The logic of these queries is provided by the BOPF.

More Information

For more information on the model guidelines on the node attribute query, see 1.1.5.6.1. For more information on how to create, change, and delete a query in the BOPF Enhancement Workbench, see 1.2.6.1.

1.3.2.13.2 Custom Query

In order to create a custom query, the implementing class of the query must be implemented. This class corresponds to a common ABAP class with the /BOBF/IF_FRW_QUERY interface maintained. If the query is created by the BOPF Enhancement Workbench, the generated class already implements this interface. For more information about the BOPF Enhancement Workbench, see 1.2.6,

The query interface consists of the following two methods

- QUERY

  This method contains the business logic of the query.

- RETRIEVE_DEFAULT_PARAM

  If the query has a parameter maintained, this method can be implemented to provide the consumer with a default parameter.

More Information

For more information on the model guidelines on custom queries, see 1.1.5.6.2. For more information on how to create, change, and delete a query in the BOPF Enhancement Workbench, see 1.2.6.1.